07｜乾坤大挪移：秒杀的削峰和限流

2021-10-08 佘志东

《手把手带你搭建秒杀系统》 课程介绍 >

讲述：佘志东
时长 15:52 大小 14.55M

你好，我是志东，欢迎和我一起从零打造秒杀系统。

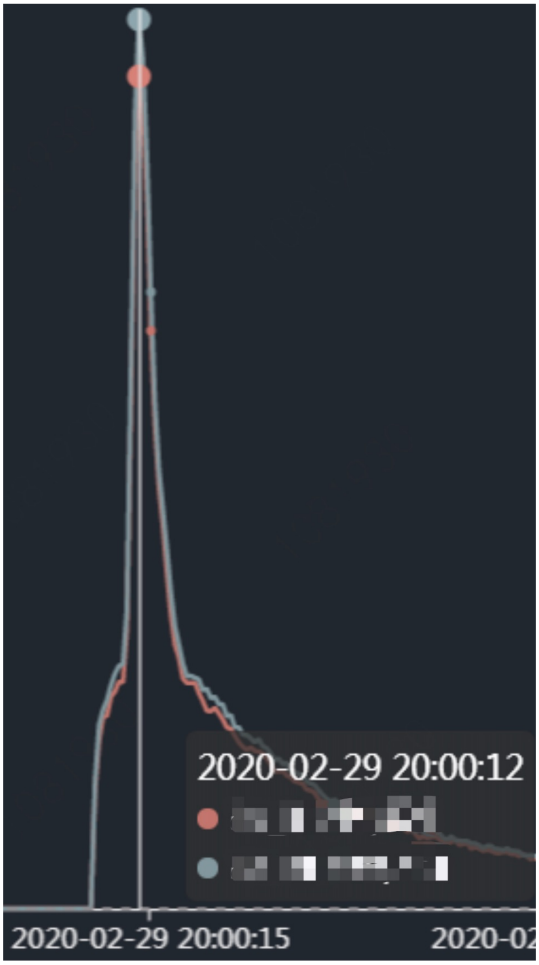
前面两节课我们讲了秒杀的隔离策略和流量控制，其目的是降低流量的相互耦合和量级，减少对系统的冲击。这节课我们将继续从**技术角度**来讨论秒杀系统的其他高可用手段——削峰和限流，通过削峰，让系统更加稳健。

削峰填谷概念一开始出现在电力行业，是调整用电负荷的一种措施，在互联网分布式高可用架构的演进过程中，也经常会采用类似的削峰填谷手段来构建稳定的系统。

削峰的方法有很多，可以通过业务手段来削峰，比如秒杀流程中设置验证码或者问答题环节；也可以通过技术手段削峰，比如采用消息队列异步化用户请求，或者采用限流漏斗对

流量进行层层过滤。削峰又分为无损和有损削峰。本质上，**限流是一种有损技术削峰；而引入验证码、问答题以及异步化消息队列可以归为无损削峰。**

我们先来看一下电商平台线上真实场景下的秒杀流量图，因为数据保密的需要，这里我隐去了具体的流量数字。但是，你可以看到这个图有个非常明显的特点，就是毛刺特别大，流量几秒内爬升到峰值，然后马上掉下来。不管是口罩、茅台，还是春运的火车票，都符合这样的流量特点。



我们现在需要做的就是通过削峰和限流，把这超大的瞬时流量平稳地承接下来，落到秒杀系统里。这就犹如武侠小说里，众人从高塔纵身跳下，张无忌运用乾坤大挪移，把对众人伤害极大的垂直自由落体运动改变为水平运动，使之安然脱险。

流量削峰

前面的章节中，我介绍过秒杀的业务特点是库存少，最终能够抢到商品的人数取决于库存数量，而参与秒杀的人越多，并发数就越高，随之无效请求也就越多。但从业务方的角度来说，肯定是希望有更多的人参与进来，点击“立即秒杀”按钮体验秒杀的乐趣。

从上面的流量监控图可以看到，在秒杀开始的时刻，会出现巨大的瞬时流量，这个流量对资源的消耗也是巨大且瞬时的。

一般来说，我们支撑秒杀系统的硬件资源是有限的，它的处理能力是恒定的，当有秒杀活动的时候，很容易繁忙导致请求处理不过来，而没有活动的时候，机器又是低负载运转。但是为了保证用户的秒杀体验，一般情况下我们的处理资源只能按照忙的时候来预估，这会导致资源的一个浪费。这就好比交通存在早高峰和晚高峰的问题，所以有了外牌限行、尾号限行等多种错峰解决方案。

因此我们需要设计一些**规则**，延缓并发请求，甚至过滤掉无效的请求，让真正可以下单的请求越少越好。总结来说，削峰的本质，一是让服务端处理变得更加平稳，二是节省服务器的机器成本。

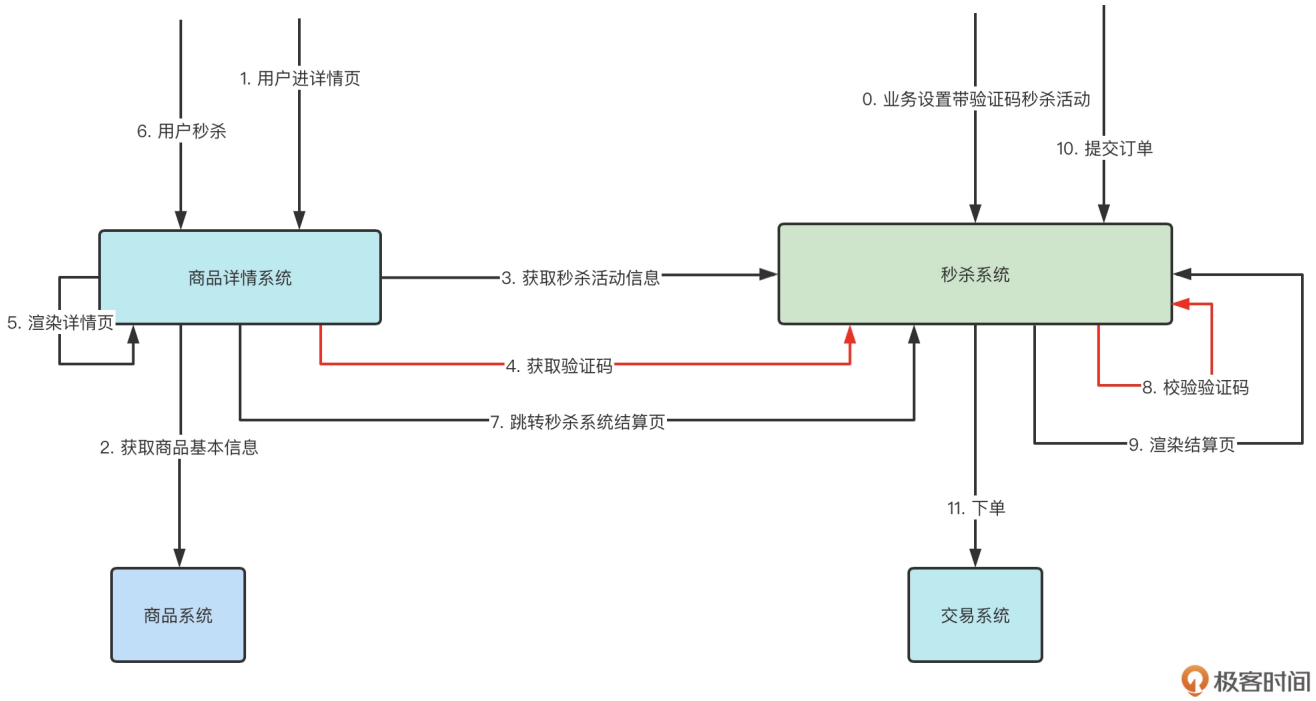
接下来，我们就重点学习几个常用的削峰手段：验证码、问答题、消息队列、分层过滤和限流。顺便看看互联网大厂里都会采用什么样的手段，以及背后的思考逻辑。

验证码和问答题

在秒杀交易流程中，引入验证码和问答题，有两个目的：一是快速拦截掉部分刷子流量，防止机器作弊，起到防刷的作用；二是平滑秒杀的毛刺请求，延缓并发，对流量进行削峰。

让用户在秒杀前输入验证码或者做问答题，不同用户的手速有快有慢，这就起到了让 1s 的瞬时流量平均到 30s 甚至 1 分钟的平滑流量中，这样就不需要堆积过多的机器应对 1s 的瞬时流量了。

以下是流程图，我来解释一下。



设计验证码流程，一般是在用户进入详情页时，先判别秒杀活动是否已经开始，如果已经开始，同时秒杀活动也配置了需要校验验证码标识，那么就需要从秒杀系统获取图片验证码，并进行渲染；用户手工输入验证码后，提交给秒杀系统进行验证码校验，如果通过就跳转至秒杀结算页。

上图增加的红线部分就是引入了验证码的秒杀流程。当然，我这里介绍的，是把验证码功能作为秒杀系统的一个模块了，而大公司一般都会有单独的验证码服务，我们不用自己造轮子，只要进行系统对接就行了。

下面我简单介绍一下验证码的实现，通过上图得知，验证码服务需提供两个基本的功能：生成验证码和校验验证码。

生成验证码，先看接口设计如下：

复制代码

```
POST /seckill/captchas.jpg?skuId=10001
```

对应的后端代码实现：

复制代码

```
    /**
```

```
     * 生成图片验证码
     */
    @RequestMapping(value="/seckill/captchas.jpg", method=RequestMethod.POST})
    @ResponseBody
    public SeckillResponse<String> genCaptchas(String skuId, HttpServletRequest
          //从cookie中取出user
          String user = getUserFromCookie(request);
          //根据skuId和user生成图片
          BufferedImage img=createCaptchas(user, skuId);
      try {
        OutputStream out=response.getOutputStream();
        ImageIO.write(img, "JPEG", out);
        out.flush();
        out.close();
        return null;
      } catch (IOException e) {
        e.printStackTrace();
        return SeckillResponse.error(ErrorMsg.SECKILL_FAIL);
      }
    }

      /**
       * 生成验证码图片方法
       */
      public BufferedImage createCaptchas(String user, String skuId) {
      int width=90;
      int height=40;
      BufferedImage img=new BufferedImage(width,height,BufferedImage.TYPE_INT_RG
      Graphics graph=img.getGraphics();
      graph.setColor(new Color(0xDCDCDC));
      graph.fillRect(0, 0, width, height);
      Random random=new Random();
      //生成验证码
      String formula=createFormula(random);
      graph.setColor(new Color(0,100,0));
      graph.setFont(new Font("Candara",Font.BOLD,24));
      //将验证码写在图片上
      graph.drawString(formula, 8, 24);
      graph.dispose();
      //计算验证码的值
      int vCode=calc(formula);
      //将计算结果保存到redis上面去，过期时间1分钟
      cacheMgr.set("CAPTCHA_"+user+"_"+skuId, vCode, 60000);
      return img;
    }

    private String createFormula(Random random) {
      private static char[]ops=new char[] {'+','-','*'};
          //生成10以内的随机数
      int num1=random.nextInt(10);
      int num2=random.nextInt(10);
      int num3=random.nextInt(10);
```

```
54      char oper1=ops[random.nextInt(3)];
55      char oper2=ops[random.nextInt(3)];
56      String exp=""+num1+oper1+num2+oper2+num3;
57      return exp;
58    }
59
60      private static int calc(String formula) {
61      try {
62        ScriptEngineManager manager=new ScriptEngineManager();
63        ScriptEngine engine=manager.getEngineByName("JavaScript");
64        return (Integer) engine.eval(formula);
65      }catch(Exception e){
66        e.printStackTrace();
67        return 0;
68      }
69    }
```

以上是自己生成图片验证码的方式，方便起见，你也可以用 Google 提供的 Kaptcha 包生成图片验证码。

同时，为了让交互更加安全，避免被篡改，我们还可以加入签名机制，后端在返回给前端图片验证码的时候，同时返回一个签名，前端在点击“抢购”按钮的时候，把用户输入的验证码以及签名提交给后端服务进行验证。这个签名可以设计如下：

复制代码

```
1 signature=base64(timestamp,md5(timestamp,vCode,skuId,user,randomSalt)
```

这里 timestamp 取生成验证码 vCode 时的时间戳，randomSalt 可以理解为后端的一个私钥。那么在前面代码的第 44 行，我们存入 Redis 的值就要换成这个 signature 了。

当前端点击“抢购”按钮时，调用后端服务如下：

复制代码

```
1 POST /seckill/settlement.html?skuId=10001&signature=ad6543audhhw13dg&timestamp
```

接下来我们看**校验验证码。**校验的逻辑比较简单，从前端的 HTTP 请求里，取得 skuId、user、signature、timestamp 和 newCode，首先验证 timestamp 是否已经过期，然后根据用户输入的验证码内容 newCode 重新计算签名 newSignature，并和 Redis 里的

signature 进行比对，比对一致表示验证码校验通过。然后我们需要删掉 Redis 的内容，避免被重复验证，这样的话一个验证码就只会被验证一次了。

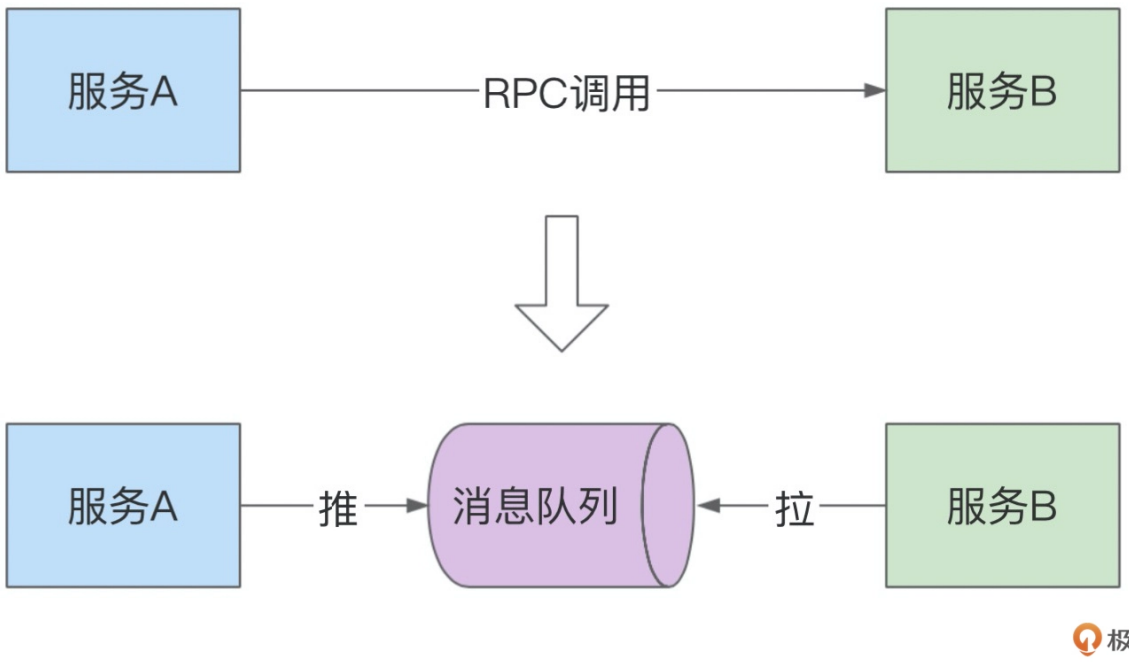
消息队列

除了验证码和问答题，另一种削峰方式是**异步消息队列**。

当服务 A 依赖服务 B 时，正常情况下服务 A 会直接通过 RPC 调用服务 B 的接口，当服务 A 调用的流量可控，且服务 B 的 TP99 和 QPS 能满足调用时，这是最简单直接的调用方式，没什么问题，目前大部分的微服务间调用也都是这样做的。

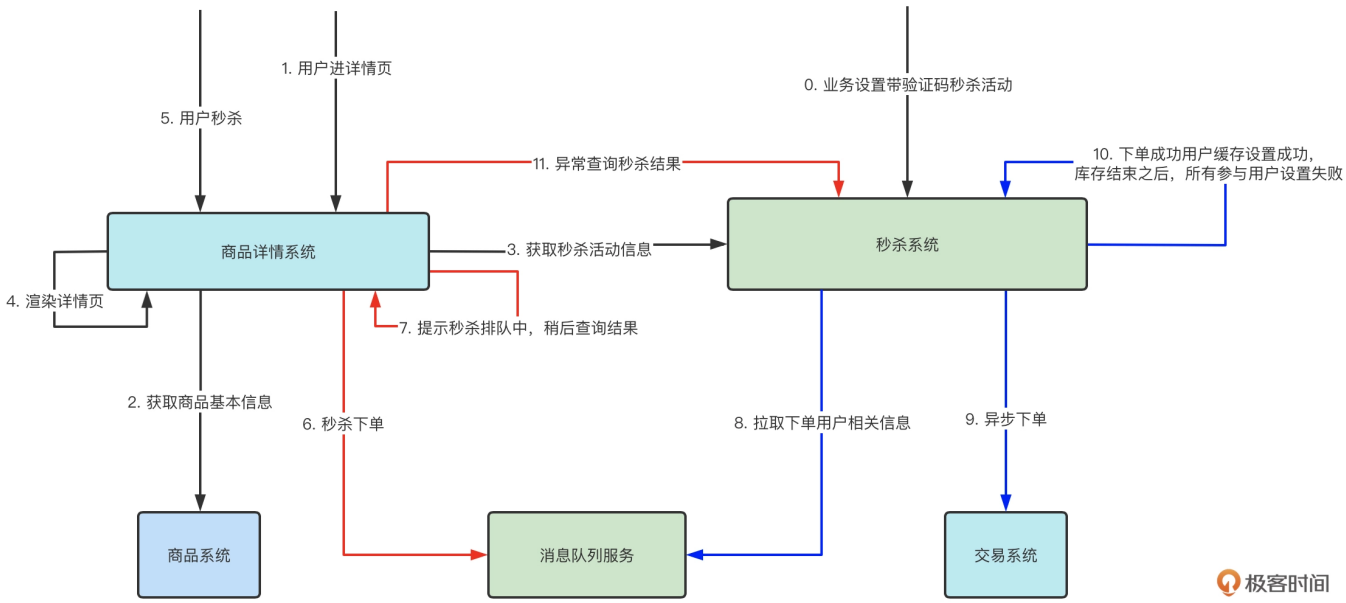
但是，试想一下，如果服务 A 的流量非常高（假设 10 万 QPS），远远大于服务 B 所能支持的能力（假设 1 万 QPS），那么服务 B 的 CPU 很快就会升高，TP99 也随之变高，最终服务 B 被服务 A 的流量冲垮。

这个时候，消息队列就派上用场了，我们把一步调用的直接紧耦合方式，通过消息队列改造成两步异步调用，让超过服务 B 范围的流量，暂存在消息队列里，由 B 根据自己的服务能力来决定处理快慢，这就是通过消息队列进行调用解耦的常见手段。



常见的开源消息队列有 Kafka、RocketMQ 和 RabbitMQ 等，大厂的基础中间件部门一般也会根据自己公司的业务特点，自研适合自己的 MQ 系统。对一般的场景来说，我推荐你用 RocketMQ，应该能解决你大部分的问题。

以上是通过 MQ 进行调用解耦的基本思路，现在我们回到秒杀的场景，看看应该怎么设计呢？请看下图：



以上红色和蓝色的部分，就是通过消息队列解耦后，详情页系统和秒杀系统各自处理的部分。因为解耦了，所以在第 6 步下单之后，其实是不知道秒杀结果的，因此在第 11 步，需要前端定期去查询秒杀结果反馈给用户。而在秒杀系统拉取消息队列进行处理的时候，也有个小技巧，那就是当前面的请求已经把库存消耗光之后，在缓存里设置占位符，让后续的请求快速失败，从而最快地进行响应。

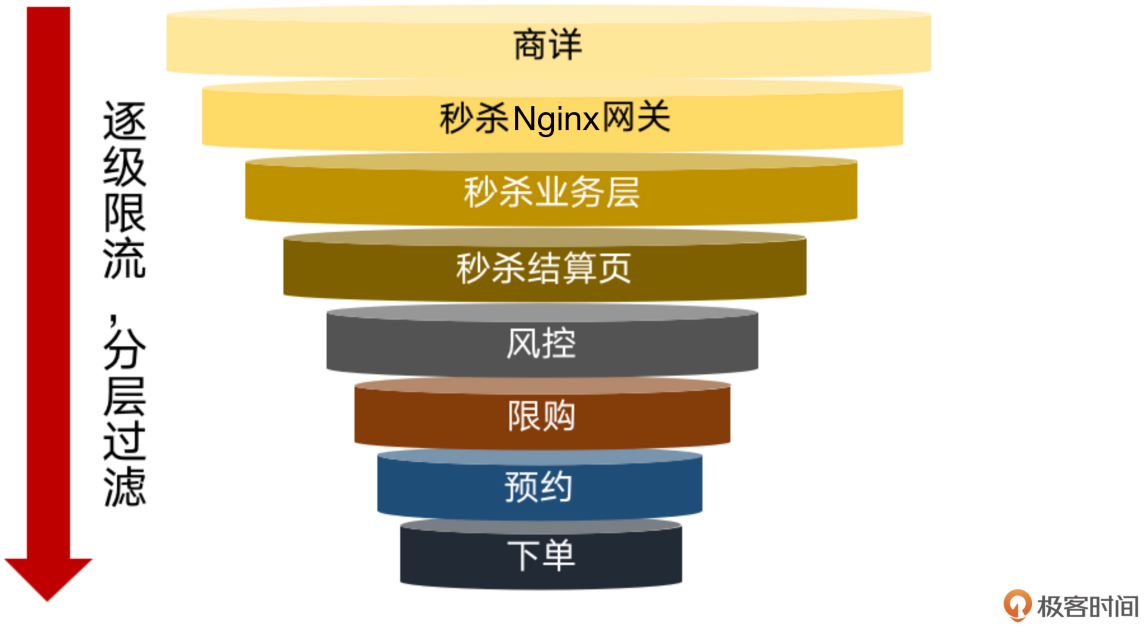
限流

削峰的方式，前面介绍了验证码 / 问答题以及消息队列，这些方式使流量峰值变得更加平滑，但也在一定程度上降低了抢购体验，容易引发用户咨询和投诉。那有没有更好的解决方式呢？接下来我们学习下限流，看看如何通过限流实现削峰。

限流是系统自我保护的最直接手段，再厉害的系统，总有所能承载的能力上限，一旦流量突破这个上限，就会引起实例宕机，进而发生系统雪崩，带来灾难性后果。

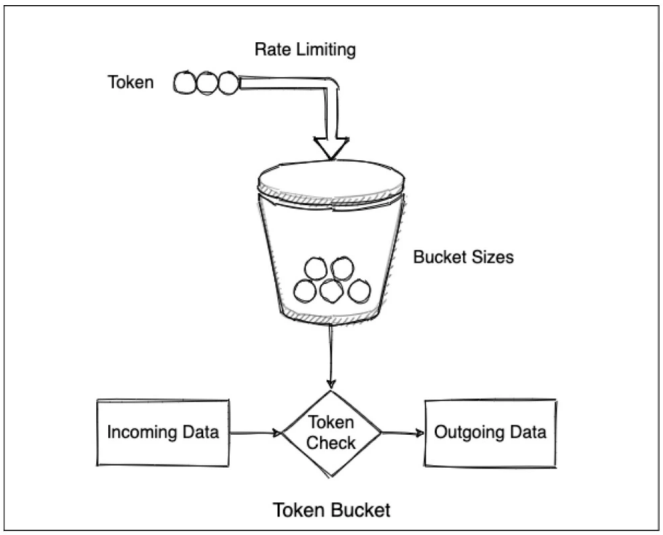
在第一讲的时候，我有和你提到过流量漏斗的概念，对于秒杀流程来说，从用户开始参与秒杀，到秒杀成功支付完成，实际上经历了很多的系统链路调用，中间有非常庞杂的系统在支撑，比如有商详、风控、登录、限购、购物车以及订单等很多交易系统。

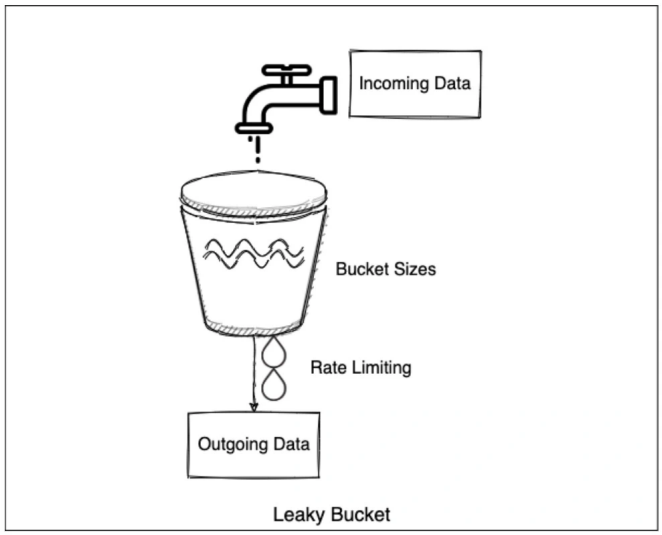
那么对于秒杀的瞬时流量，如果不加筛选，不做限制，直接把流量传递给下游各个系统，对整个交易系统都是非常大的挑战，也是很大的资源浪费，所以主流的做法是从上游开始，对流量进行逐级限流，分层过滤，优质的有效的流量最终才能参与下单。



这是系统的**流量漏斗示意图**，通过风控和防刷筛选刷子流量，通过限购和预约校验过滤无效流量，通过限流丢弃多余流量，最终秒杀系统给到下游的流量就是非常优质且少量的了。

限流常用的算法有令牌桶和漏桶，有关这两个算法的专业介绍，你可以参考：https://hansliu.com/posts/2020/11/what-is-token-bucket-and-leaky-bucket-algorithms.html





下面我们针对 demo-nginx 和 demo-web 两个应用，介绍一下具体的限流方法。

demo-nginx 网关限流

先开始**准备工作**，俗话说，工欲善其事必先利其器，在开发之前，我们先把 Nginx 的日志给配置起来，方便我们后续开发的调试与验证。

Nginx 日志配置：Nginx 主要有两种类型的日志文件。

一个是 error_log，用来记录我们的系统日志，以及主动打印的业务日志，配置语法为：

复制代码

```
error_log <日志文件路径> <日志级别>;
```

另一个是 access_log，这个日志主要用来记录我们的请求和返回相关的信息，配置语法为：

复制代码

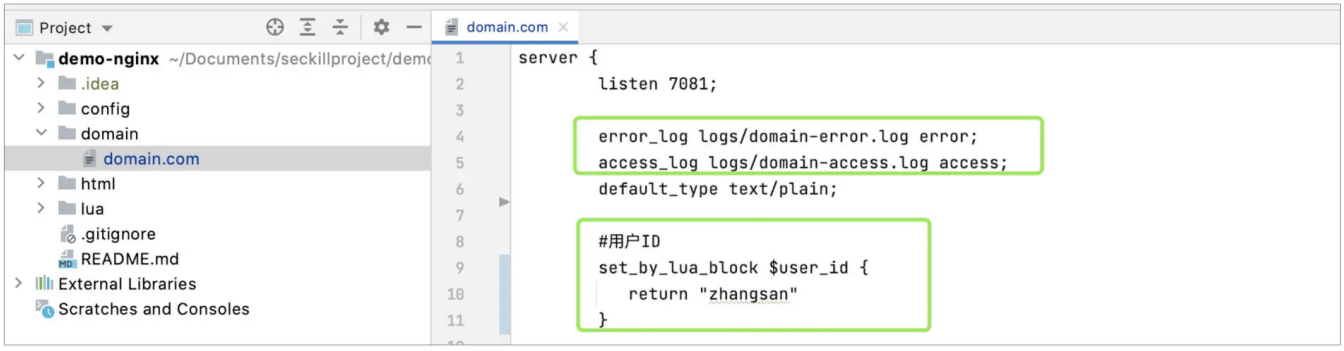
```
access_log <日志文件路径> <日志格式定义的名称>;
```

如果想自定义输出日志格式，需要使用 log_format 来实现，下面我们就来配置一下这两种日志。

首先我们在 nginx.conf 中定义一个名为 access 的日志格式，如下图所示：

```
worker_processes 1;

error_log logs/error.log error;

events {
    worker_connections 256;
}

http {


    log_format  access  '$remote_addr - $remote_user [$time_local] "$request" $status '
        '"$upstream_addr" "$upstream_status" "$upstream_response_time" userId:"$user_id"';

    lua_package_path "/Users/wangzhangfei5/Documents/seckillproject/demo-nginx/lua/?.lua;;";

    include /Users/wangzhangfei5/Documents/seckillproject/demo-nginx/domain/domain.com;
    include /Users/wangzhangfei5/Documents/seckillproject/demo-nginx/config/upstream.conf;
}
```

然后在 domain.com 中配置 error_log 和 access_log，如下图所示：



这里我使用了内置变量以及自定义变量 $user_id（通过 set_by_lua_block 定义并赋值）。现在我们已经把日志配置好了，接下来就开始今天的重头戏吧，就从 Nginx 限流开始讲起。

这里的 Nginx 限流，主要是依赖 Nginx 自带的限流功能，针对请求的来源 IP 或者自定义的一个关键参数来做限流，比如用户 ID。其配置限流规则的语法为：

复制代码

```
limit_req_zone <变量名> zone=<限流规则名称>:<内存大小> rate=<速率阈值>r/s;
```

解释一下：

以上 limit_req_zone 是关键字，< 变量名 > 是指定根据什么来限流；

zone 是关键字，< 限流规则名称 > 是定义规则名称，后续代码中可以指定使用哪个规则；

< 内存大小 > 是指声明多大内存来支撑限流的功能；

rate 是关键字，可以指定限流的阈值，单位 r/s 意为每秒允许通过的请求，这个算法是使用令牌漏桶的思想来实现的。

那么明白了语法之后，下面我们就**动手定义一个限流规则**，看看实际效果。

复制代码

```
http {
    limit_req_zone $binary_remote_addr zone=one:10m rate=1r/s;
    server {
```

```
        location /search/ {
            limit_req zone=one burst=2 nodelay;
        }
    }
}
```

以上是基于 IP 地址进行限流的例子，你可以根据实际的情况调整 rate 和 burst 的值，在秒杀的场景下，一般我们会把 rate 和 burst 设置的很低，可以都为 1，即要求 1 个 IP1 秒内只能访问 1 次。

但根据 IP 地址设置限流时要慎重，会存在误杀的情况，特别像公司内用户，他们的出口 IP 就那么几个，很容易就触发了限流，所以我一般在参与阿里、苏宁或京东的秒杀活动时，都会切换到 4G 网络，避免用公司网络。

除了基于 IP 限流外，我们还可以设计基于用户的 userId 进行限流。

复制代码

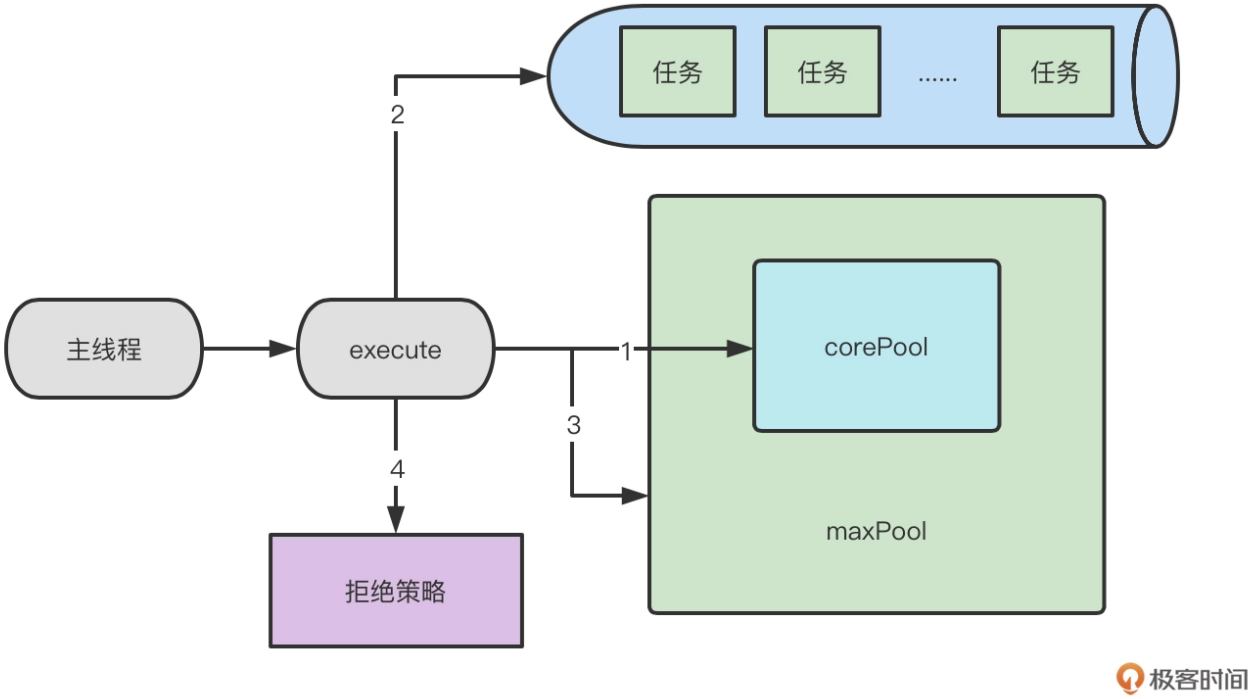
```
limit_req_zone $user_id zone=limit_by_user:10m rate=1r/s;
```

demo-web 应用层限流

以上是 Nginx 网关层的限流，接下来我们进入应用层的限流。应用层的限流手段也是比较多的，这里我们重点介绍通过线程池和 API 限流的方法。

线程池限流

Java 原生的线程池原理相信你非常清楚，我们可以通过自定义线程池，配置最大连接数，以请求处理队列长度以及拒绝策略等参数来达到限流的目的。当处理队列满，而且最大线程都在处理时，多余的请求就会被拒绝策略丢弃，也就是被限流了。



API 限流

上面介绍的线程池限流可以看做是一种并发数限流，对于并发数限流来说，实际上服务提供的 QPS 能力是和后端处理的响应时长有关系的，在并发数恒定的情况下，TP99 越低，QPS 就越高。

然而大部分情况是，我们希望根据 QPS 多少来进行限流，这时就不能用线程池策略了。不过，我们可以用 Google 提供的 RateLimiter 开源包，自己手写一个基于令牌桶的限流注解和实现，在业务 API 代码里使用。当然了，大厂中都会有通用的限流机制，你直接用就行了。

复制代码

```
/**
 * 自定义注解  限流
 */

@Target({ElementType.PARAMETER, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface MyRateLimit {
     String description() default "";
}
```

我们自定义一个切面：

复制代码

```
/**
 * 限流 AOP
 */

@Component
@Scope
@Aspect
public class LimitAspect {
    //引用RateLimiter，内部是基于令牌桶实现的
    private static RateLimiter rateLimiter = RateLimiter.create(100.0);

    //定义限流注解的pointcut
    @Pointcut("@annotation(com.ecommerce.seckill.aop.MyRateLimit)")
    public void MyRateLimitAspect() {
    }

    @Around("MyRateLimitAspect()")
    public  Object around(ProceedingJoinPoint joinPoint) {
        Boolean flag = rateLimiter.tryAcquire();
        Object obj = null;
        try {
            if(flag){
                obj = joinPoint.proceed();
            }
        } catch (Throwable e) {
            e.printStackTrace();
        }
        return obj;
    }
}
```

业务层代码实现：

复制代码

```
@Override
@MyRateLimit
@Transactional
public SeckillResponse initData(String skuId, String userName) {
    //此次为业务代码实现
}
```

小结

这节课我们介绍了削峰的多种手段，有验证码、问答题、消息队列以及限流等。实际上，这些削峰的方式都可以达到控制流量的目的，你可以根据自己的情况进行选择。

验证码是一种非常常见的防刷手段，大多数网站的登录模块中，为避免被机器人刷，都会加入图片验证码。而在秒杀系统中，我们除了用验证码来防刷外，还有一个目的就是通过验证码进行削峰，达到流量整形的目的。除了图片验证码，你一定也见过短信和语音验证码，那为什么在秒杀系统的削峰中，我们通常会选择图片验证码呢？主要还是出于成本和用户体验的考虑。

消息队列是常用的应用解耦方式，通过把同步调用改造成异步消息，消费方可以根据自己的能力来处理请求，而不用担心被瞬时流量打垮。当然了，如果库存已经卖完，那么消费方在处理请求的时候，可以快速失败，这样也不用担心消息的长期积压。

最后，我介绍了几种限流的方式，和其他削峰方式相比，限流是有损的。限流实际上是根据服务自身的容量，无差别地丢弃多余流量，对于被丢弃的流量来说，这块的体验是受损的。另外，因为秒杀流量会经历很多交易系统，所以我们在设计时需要从起始流量开始，分层过滤，逐级限流，这样流量在最后的下单环节就是少量而可控的了。

另外，在这一节课开始的时候，我有留下一个小伏笔，就是在介绍了各种削峰手段后，互联网大厂在实践中一般都是怎么选择的呢？其实，如果你去体验下像天猫、京东的秒杀，就会发现他们总体是比较类似的，基本不会使用验证码或答题这两种方式，因为对于头部电商平台来说，体验可能不是那么友好。他们比较偏向采用非公平的抢购策略，也就是有损的逐级限流和分层过滤，背后最重要的考虑其实就是兼顾了体验与系统资源。

思考题

我们在介绍削峰手段的时候，有提到过问答题也是一种削峰方式，但是在这一讲中，因为不常用的关系，我略去了问答题的设计。这里我想请你思考一下，如果让你来设计秒杀的问答题系统，将其作为一种削峰方式，你会怎么设计呢？

以上就是这节课的全部内容，欢迎你在评论区和我讨论问题，交流经验！

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 2 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 06 | 谋定后动：秒杀的流量管控

下一篇 加餐 | 高并发场景：如何提升对突发事件的应急处理能力？

精选留言 (3)

写留言

董永政 置顶

2021-10-08

给的网址访问不了

展开

编辑回复: 同学你好，我刚试了下可以正常访问，麻烦你检查下网络哈～有问题可以再反馈！

GitHub 地址：https://github.com/sanyecao-seckill

威

2021-10-08

如果是分布式限流，一般会怎样做呢

展开

Geek_d0863b

2021-10-08

老师 您能尽快更新github上的代码嘛😄 谢谢了！

展开

编辑回复: 收到，加班加点更新中～感谢耐心等待！